A Java Library for Bidirectional XML Transformation

Dongxi Liu Zhenjiang Hu Masato Takeichi
Kazuhiko Kakehi Hao Wang

We propose a Java library `BiXJ` for bidirectional XML transformation. A bidirectional transformation generates target XML documents from source XML documents in forward transformations, and updates source documents by reflecting back modifications on target documents in backward transformations. The benefit of using `BiXJ` is that users can get the corresponding backward transformation automatically just by writing one forward transformation. `BiXJ` has addressed several limitations of the existing bidirectional transformation languages, and can be used for general purpose XML processing. For example, this library provides a way to write bidirectional XPath expressions, which is widely used to locate and extract data from XML documents. To validate the usability and expressiveness of `BiXJ`, we have bidirectionalized some typical examples of XQuery and XSLT using this library. The results of these experiments are promising.

1 Introduction

XML is widely used as the de facto standard format of data exchange or repository. XML documents often need to be transformed for different reasons. For example, an XML file is transformed into HTML format for displaying in Web browsers, or transformed into a small XML file containing only interesting data for users. In some cases, the target XML documents generated by transformations are probably modified by users to update some data or to correct some errors, and it is desirable that these modifications on the target documents can be reflected back into source documents.

However, the current popular XML transformation languages, such as XSLT [1] and XQuery [2], perform transformation only in one direction, generating target documents from source documents. As a result, modifications on target documents cannot be reflected back into the source documents without using other separate mechanisms.

This work presents a Java library `BiXJ` for bidirectional XML transformation. A bidirectional transformation program can be executed in two directions: forward direction and backward direction. The forward transformation transforms a source XML document into a target document, while the backward transformation transforms the target document (probably modified) together with the original source document into the updated source document. After the backward transformation, the modifications in the target document will be reflected back into the source document. When writing `BiXJ` transformations, users just need to consider how to generate the expected target documents from source documents, which is a similar programming paradigm as that of using XSLT and XQuery. With `BiXJ`, no extra efforts or other separate mechanisms are needed for users to update source documents after target documents are modified. The only work they need to do is to execute their `BiXJ` programs backwardly.

双方向 XML 変換のための Java ライブラリ.

劉 東喜, 胡 振江, 武市 正人, 東京大学 大学院情報理工学系研究科 数理情報学専攻, Department of Mathematical Informatics, The University of Tokyo.

筧 一彦, 東京大学産学連携本部, Division of University Corporate Relations, The University of Tokyo.

王 浩, 東京大学 大学院情報理工学系研究科 創造情報学専攻, Department of Creative Informatics, The University of Tokyo.

コンピュータソフトウェア, Vol.XX, No.X (200X), pp.XX–XX.

[論文] 2005 年 yy 月 zz 日受付.

In BiXJ, each language construct is defined with two meanings: one for forward transformations and the other for backward transformations. This style of bidirectional transformation techniques has been proposed in the literature [3][4]. However, the languages in [3][4] are both domain-specific. The language in [3] is designed for synchronizing tree-structured data, and the language in [4] is mainly used in an editor for editing tree-structured data. Because of their domain-specific purposes, these languages have several limitations when they are used as general purpose XML processing languages.

First, these languages do not take the standard XML data model [5], which is a tree with ordered labeled nodes or non-labeled data nodes (i.e., text content). The language in [3] takes a tree-structured data model that only allows unordered labeled nodes without repeated labels, and the language in [4] uses a model which does not allow non-labeled data nodes.

Second, a transformation in these languages can only generate one target XML element, which is too restrictive for XML transformations. For example, if a `book` element contains one `title` element and more than one `author` elements, then extracting the author information from this `book` element will return a sequence of `author` elements. XQuery and XSLT allow to return more than one elements in their transformation results.

Third, these languages use their own specific methods to destruct tree-structured data, which are probably unnatural for users who have been familiar with the existing XML transformation languages. For example, the construct `hoist` in [3] and `hoistX` in [4] return a child element if the source document contains only this element as its child. However, in XML transformation, a widely accepted way is to use XPath [6] to locate and extract data from XML documents, which is used in both XQuery and XSLT.

Fourth, the view updating semantics of these languages is too restrictive. This semantics gives conditions on whether a bidirectional transformation is well-behaved [3]. That is, whether it can correctly reflect modifications in target documents back into source documents. Under this restrictive semantics, to guarantee the well-behavedness of transformations written in these languages, some reasonable modifications on target data or the expressiveness of transformation languages have to be restricted. This problem is discussed more in Section 4.

In this work, BiXJ is designed to extend the expressiveness and usability of these existing languages, so that it can be used for general purpose XML processing. In order to design such library, we have to address the first three limitations discussed above. However, we are in a dilemma because the fourth limitation does not allow us to make too much extensions, otherwise the transformations written in the extended language are probably not well-behaved. Our approach in this work is to seek a more flexible view updating semantics for bidirectional transformations, and then under this semantics to extend the existing languages. To demonstrate its expressiveness and usability, we have used BiXJ to bidirectionalize some typical examples of XQuery and XSLT.

Since this library is written in Java, it can be applied in any case where Java is used to process XML documents, and moreover bidirectional transformations are expected. For example, Java is one of the most popular languages to implement web service [7], so with this library, a Java program can not only provide interesting XML data for clients, but also update the original source XML data on web servers after receiving the modified data from clients. On the other hand, the transformation constructs in this library can also be represented as XML elements and then interpreted using the corresponding Java classes at runtime. This provides a convenient way for users to write BiXJ programs if they do not know Java or their programs are not used together with Java programs. The Java interface of BiXJ and its XML representation will be described in detail in Section 3.

The main contribution of this paper is the design and implementation of a Java library BiXJ for bidirectional XML transformation, which addresses the limitations of the existing bidirectional transformation languages from the following aspects:

- BiXJ uses the standard XML data model and allows to construct or destruct XML documents in a similar way as that in current XML transformation languages. For example, the `child` and `descendant` axes of XPath are supported in BiXJ, and with them we have demonstrated how to make XPath expressions bidi-

rectional.

- We define a more flexible semantics of bidirectional transformations, which underlies the well-behavedness of `BiXJ` transformations without restricting its expressiveness and reasonable modifications on target documents.
- The transformations written in `BiXJ` can be updated after backward execution because some modifications on target documents should be reflected back into transformations rather than into source documents.
- `BiXJ` provides flexible ways to write transformations, i.e., in Java class or in XML format, and it is expressive enough to bidirectionalize typical use cases of XQuery and XSLT.

The remainder of this paper is organized as follows. Section 2 gives a practical scenario of using the bidirectional XML transformation. Section 3 overviews the library `BiXJ`. Section 4 discusses some issues in designing `BiXJ`. Section 5 describes the transformations in the library in detail. Section 6 gives examples of bidirectionalizing typical XQuery and XSLT transformations. Section 7 discusses the related work and Section 8 concludes the paper and gives the future work.

2 Bidirectional Transformation in Use

Consider the following XML fragment, which contains some book information. The top element, tagged by `books`, contains three child `book` elements. Each `book` element contains the child elements `title`, `author`, `year` and `publisher`.

```
<books>
  <book>
    <title>Computer Programming</title>
    <author>Tom</author><year>2003</year>
    <publisher>Now Century</publisher>
  </book>
  <book>
    <title>Data Structure</title>
    <author>Peter</author><year>2005</year>
    <publisher>Great Press</publsher>
  </book>
  <book>
    <title>Computer Graphecs</title>
    <author>Tom</author><year>1999</year>
    <publisher>ACM Press</publisher>
  </book>
</books>
```

Suppose that this XML is stored on a web server. The web service on this server allows authors to select the books they wrote for checking and correcting error information, or ordinary customers to request the information of their interesting books. For the request from authors, the server performs forward transformations and returns elements with the tag `mybooks`, while for the request of ordinary customers it generates elements tagged by `interestingbooks`.

For example, when the author Tom wants the title and publisher information of his books, the server should transform the above source document into the following one:

```
<mybooks>
  <book>
    <title>Computer Programming</title>
    <publisher>Now Century</publisher>
  </book>
  <book>
    <title>Computer Graphecs</title>
    <publisher>ACM Press</publisher>
  </book>
</mybooks>
```

Unfortunately, Tom finds several errors in the above document: “Now” in the `publisher` element of the first book should be “New” , and “Graphecs” in the `title` element of the second book should be “Graphics”. After correcting these errors, Tom sends this changed document back to the server and the server will perform a backward transformation to update the source document.

Later, a customer asks the server to list the title, author and publisher information of the books published before the year 2000. Then, after a forward transformation, the server answers with the following XML data:

```
<mybooks>
  <book>
    <title>Computer Graphics</title>
    <author>Tom</author>
    <publisher>ACM Press</publisher>
  </book>
</mybooks>
```

Note that this document contains the correct book title now. This scenario is depicted in Figure 1.

3 Overview of BiXJ

Our library is built on JDOM [8], which provides an easy way to process XML document in Java. The class `Element` in JDOM abstracts the elements in XML documents and operations on them. In this

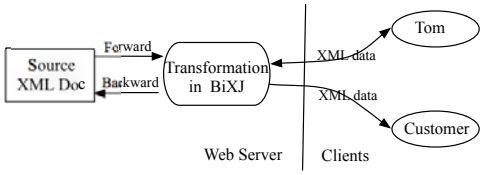

Fig. 1 A Scenario of Bidirectional Transformation

work, we refine the class `Element` into three subclasses `SrcElement`, `TgtElement` and `CodeElement` to help users to understand the roles of elements in transformation. The first two subclasses correspond to the source and target elements, respectively, and the third subclass will be introduced later.

The interface of `BiXJ` transformation is defined as in Figure 2. In this interface, the methods `tranForward` and `tranBackward` perform the forward and backward transformations, respectively. However, unlike the corresponding *get* and *put* in [3], these two methods take lists of elements as their source and target data. In XML transformation, it is more interesting that the target data is a list, but when we compose two transformations, the first transformation may return a list of elements, and the second takes this list as its input, so the source data in these methods is also given a list type.

The method `dump` is specific to this library. For a runtime object of type `XAction`, this method returns an XML fragment of type `CodeElement` that represents this transformation object. In `BiXJ`, transformations can be updated during backward executions, where the state of transformation objects (i.e. some fields) are changed. The `dump` method is used to output the updated transformations to users. In Section 5, we will see some constructs in `BiXJ` that update transformations in backward executions, such as `XConst`. In addition, the class `CodeElement` has the method used to interpret a code element as a transformation object during runtime.

We provide two ways to represent BiXJ transformation in Java class or by code element. Our design purpose is that Java class will be used by Java programmers when writing Java programs to implement bidirectional XML transformations, and code element will be used by users who do not know Java or their applications are not Java applications. We can imagine the following interesting applications for the second representation:

- Code elements can be used as the intermediate code when translating the expressions of high level transformation languages into bidirectional transformations in `BiXJ`. The XML representation is more compact, so the translation result is more readable. In the examples of Section 6, code elements are used to describe the typical transformations of XQuery and XSLT.
- Code elements provide a means of modifying transformation objects at runtime. After dumping a transformation object into a code element, this element will be processed like an ordinary XML element, and then it can be interpreted as a new transformation object again. This technique can help implement self-adjusting transformations. As we will see, the implementation of `xmap` uses this technique to update its argument transformation.
- Code elements can be incorporated into XML documents to construct Programmable Structured Documents (PSD) [9]. For example, suppose that there is a book document containing a child element for the table of contents and more than one child elements for chapters, and the table of contents consists of all chapter titles. In the PSD framework, we do not need to write the title elements in the table of contents explicitly, instead we can write an expression as the content of the table of contents to compute these titles from the chapters. In this example, we can use code elements of `BiXJ` to write this expression, and when the titles in the table of contents are changed, the corresponding titles in chapters can also be updated, which is more flexible than writing this expression in Haskell [10].
- Code elements can be used as a kind of mobile code. For example, in a cluster of web servers, a runtime transformation object in one server can be dumped and moved to other machine, and then interpreted and run again.

All classes in `BiXJ` are summarized in Figure 4, where x denotes a transformation object, and c is

```
public interface XAction{
    public List<TgtElement> tranForward(List<SrcElement>  sd);
    public List<SrcElement> tranBackward(List<SrcElement>  sd, List<TgtElement>  td);
    public CodeElement dump();
  }
```

Fig. 2 The Java Interface of BiXJ

```
<xseq>
  <xchildrennm>book</xchildrennm>
  <xmap>
    <xif>
      <xequals><path>1<path>
        <value>Tom</value>
      <xequals>
      <xid /><xhide />
    <xif>
  </xmap>
  <xmap>
    <xseq>
      <xdistribute>2</xdistribute>
      <xnewroot>book</xnewroot>
      <xzip>
        <xchildrennm>title</xchildrennm>
        <xchildrennm>publisher</xchildrennm>
      </xzip>
    </xseq>
  </xmap>
  <xnewroot>mybooks</xnewroot>
</xseq>
```

Fig. 3 The Transformation for Tom

its corresponding code element; *pred* represents an object for conditions and *cpred* is its XML representation. The formal definitions of these classes will be given in Section 5. With code elements, the transformation for Tom in last section is given in Figure 3. The element `xequals` represents a predicate, which tests (in this case) whether the author under `book` element is Tom. Its parameter `<path>1</path>` indicates that the content of the second child element in the source document is tested.

4 Design Issues

Any transformation in `BiXJ` can be executed in two directions: forward or backward. For forward transformations, we care about the expressiveness of `BiXJ`, while for backward transformations, we need to concern its view updating semantics, which is to determine whether a backward transformation correctly reflects modifications in target documents back into source documents. In this section, we discuss the view updating semantics for bidirectional transformations in `BiXJ`, and in next section, we will give its detailed definition.

4.1 View Updating Semantics

In [3], the well-behavedness of bidirectional transformations is guaranteed by two properties: the first one, characterized by the `GETPUT` law, says the backward transformation of an unchanged target document should not change the source document, that is, a well-behaved bidirectional transformation is side-effect free; the second one, characterized by the `PUTGET` law, says the transformation of an updated source document should get the same modified target documents, that is, a well-behaved bidirectional transformation should reflect all modifications in the target document into the source document.

However, the second property is found restrictive for `BiXJ`. In what follows, we first characterize the first property in `BiXJ` transformations and then discuss the reasons why the second property is restrictive.

With methods `tranForward` and `tranBackward`, the property of side-effect free can be stated as follows, which is the `GETPUT` law in [3]:

$x.\texttt{tranBackward}(sd, x.\texttt{tranForward}(sd)) = sd$

where sd is the source data, and x is a transformation object having interface `XAction`. This formula means, if we transform sd into the target data, and then transform it back immediately, then the obtained source document should still be sd.

The property stated by `PUTGET` law in [3] can be described as follows in our work:

$x.\texttt{tranForward}(x.\texttt{tranBackward}(sd, td)) = td$

Class Constructors	Code Element	Description (Forward Meaning)
XSeq([x_1,...,x_n])	<xseq>$c_1 \ldots c_n$</xseq>	Applies x_1,...,x_n sequentially.
XMap(x')	<xmap>c'</xmap>	Applies x' to each element in the source data.
XZip([x_1,...,x_n])	<xzip>$c_1 \ldots c_n$</xzip>	Applies x_1,...,$x_n$to the corresponding child of the source data, which must be an element.
XIf($pred$, x_1, x_2)	<xif>$cpred$ c_1 c_2</xif>	Applies x_1 to the source data if this data satisfies $pred$, otherwise applies x_2.
XID()	<xid />	Identity transformation.
XConst($elmobj$)	<xconst>elm</xconst>	Constant transformation with $elmobj$ as its result.
XHide()	<xhide />	Returns the empty value ().
XModifyName(nm)	<xmodifyname>nm </xmodifyname>	Modifies the tag of the source element to nm.
XNewRoot(nm)	<xnewroot>nm</xnewroot>	Makes the source data as the content of a node with tag nm.
XDistribute(n)	<xdistribute>n </xdistribute>	Makes n copies of the source element.
XChildren()	<xchildren />	Returns all child elements of the source element.
XDescendant()	<xdescendant />	Returns all descendant elements of the source element.
XChildrenNm(nm)	<xchildrennm>nm </xchildrennm>	Returns all child elements with name nm in the source element.
XActionNFun()		Used as the parent class for all non-invertible transformations.

Fig. 4 Classes in BiXJ

where *sd* is the source data and *td* is the target data obtained by modifying the original target data generated by x.`tranForward`(sd). However, such property is too restrictive. We will give four cases where this property is violated.

The first case is relevant to conditional transformations, such as `xif` in Figure 3. For the example in Section 2, if Tom modifies all his names in the target data into his full name "Tom Bill", then the forward transformation x.`tranForward` will produce nothing from the updated source data because there is no book with the author "Tom". Hence, the above property is violated, though the modifications have been correctly reflected into *sd*.

The second case is that sometimes the modifications on target documents should be reflected back into the transformations rather than into the source documents. Still using the example in Section 2, if Tom changes the tag `mybooks` in the target data into `Books-of-Tom`, the source data should be kept unchanged. Rather, the transformation, < `xnewroot` >`mybooks`< /`xnewroot` >, in Figure 3 should be updated as <`xnewroot`>`Books-of-Tom`< /`xnewroot`>, so that Tom can see his modifications really take place when he does forward transformation again.

In the third case, the violation is caused by data dependency in the target data, which has been recognized in [4]. For example, if an element in a source document is duplicated in the target document, then these two replicas are mutually dependent, and only modifying one replica will cause the `PUTGET` law violated. However, the proposed laws in that work, `PUTGETPUT` and `GETPUTGET`, is not strong enough to guarantee that all reasonable modifications can be reflected back into the source data in a well-behaved transformation.

The fourth case is relevant to non-inverftible functions. For example, suppose there is a transformation that sums two integers in source data in the forward direction and keeps the source data unchanged in the backward direction. In this case, changing the sum in the target data will violate the `PUTGET` law since this modification is abandoned in the backward transformation.

A way of solving the above problems is to restrict possible modifications on target documents or exclude those problematic language constructs. For example, the string "Tom", the tag `mybooks` and the sum in the above examples are not allowed to change and the duplication operation (i.e., `xdistribute` in `BiXJ` and `Dup` in [4]) should be ex-

cluded. Obviously, this approach is too restrictive.

In our new view updating semantic, we still restrict modifications, but we only restrict modifications on the result of non-invertible functions, such as sum of two integers. The definition of our view updating semantics depends on the differences between two documents with the same structure, such as the differences between the original source document and the corresponding updated source document. The differences between two documents are represented as a multiset of pairs, and each pair includes two different text strings, which are either tag names or text contents. A pair represents a modification, that is, its first component is changed to the second one. We write `diff`(od,md) for the differences between the original document od and its modified document md. For the example in Section 2, suppose sd is the original source document on the web server and sd' is the updated source document. Then `diff`(sd,sd') ={("Now Century", "New Century"), ("Computer Graphecs","Computer Graphics")}. Since transformations in `BiXJ` can be represented as XML documents, the differences between two documents can also be applied to two transformations.

The view updating semantics in this work is defined as follows: Suppose sd is a source document, x a transformation object having interface `XAction`, td a target document of sd transformed by x (i.e., x.`tranForward`(sd) = td), and td' is obtained by modifying td and the modified data does not include the results of non-invertible functions. Then the transformation x is well-behaved if the following condition holds:

$$\texttt{diff}(sd, sd') \cup \texttt{diff}(c, c') = \texttt{diff}(td, td')$$

where $sd' = x.\texttt{tranBackward}(sd, td')$, x' is x with its state updated by the backward transformation, and c and c' are the XML representations of x and x', respectively.

A sharp reader may argue that if there are conflict modifications in td', the above equation will not hold because not all modifications can be reflected back successfully. For the example in Section 2, there are two books written by Tom in the target data, and if Tom changes the tag `mybooks` in his first book into `Books-of-Tom`, and that in his second book into `Tom-Book`, then these two modifications cause conflict when updating the parameter of `xnewroot` and one of the modifications has to be abandoned. Note that the above equation depends on the normal terminated execution of `tranBackward`. In `BiXJ`, some runtime technique helps solve this problem. If there are conflict modifications on target data, the execution of `tranBackward` will be aborted due to exceptions raised in `xdistribute` or `xmap`. It can be checked that the above equation holds for all `BiXJ` transformations.

5 The Implementation of BiXJ

In this section, we will give the detailed definition of `BiXJ` transformations listed in Figure 4. The transformations in `BiXJ` can be divided into two kinds: basic transformations and transformation combinators. Some other interesting transformations can be derived from these basic transformations and transformation combinators. As examples of derived transformations, we will demonstrate how to make XPath expressions bidirectional.

The syntax of XML elements is defined below, where the ending tags of elements are omitted and their contents are put into brackets to save space.

$$\begin{aligned} element &::= \ <tag>\ [element, ..., element] \\ &\quad\ \mid\ <tag>\ [string] \mid () \\ tag &::= \ string \end{aligned}$$

Note that `()` is a special element, and in implementation, it is just a *null* element reference. We will use x for denoting a transformation object, sd, td or d for a sequence of elements, and e for a single element.

5.1 Basic Transformations

A basic transformation generally performs one particular operation on source documents, such as the constant transformation or the transformation returning the child elements of source data.

`XID`: Let x = `new XID`().

$$\begin{aligned} x.\texttt{tranForward}(sd) &= sd \\ x.\texttt{tranBackward}(sd, td) &= td \end{aligned}$$

`XID` is the identity transformation, which always keeps the source and the target data identical.

`XConst`: Let x = `new XConst`(e).

$$\begin{aligned} x.\texttt{tranForward}(sd) &= e \\ x.\texttt{tranBackward}(sd, e') &= sd \end{aligned}$$

In this transformation, the state of the object x

is changed with its parameter e replaced by e' after a backward execution. This effect is implicit in this definition, but it can be observed when we use the method `dump` to output x in its XML format. When this updated x is used to perform a forward transformation, the target data will be e' rather than e. In this transformation and some following transformations, the element e should be understood as a singleton list $[e]$. We omit the brackets for brevity.

`XHide`: Let $x =$ `new XHide`().

$$\begin{aligned} x.\texttt{tranForward}(sd) &= () \\ x.\texttt{tranBackward}(sd, ()) &= sd \end{aligned}$$

This transformation is to hide the source data, so its target data is the empty value ().

`XModifyName`: Let $x =$ `new XModifyName`(nm) and $e = <tag>[d]$.

$$\begin{aligned} x.\texttt{tranForward}(e) &= <nm>[d] \\ x.\texttt{tranBackward}(e, <nm'>[d']) &= <tag>[d'] \end{aligned}$$

After backward execution, the state of x is changed with its parameter nm replaced by nm' and the contents of the source data is also updated. This transformation takes a single element as its source data. It can be passed to the transformation combinator `XMap` to transform an element list.

`XNewRoot`: Let $x =$ `new XNewRoot`(nm).

$$\begin{aligned} x.\texttt{tranForward}(sd) &= <nm>[sd] \\ x.\texttt{tranBackward}(sd, <nm'>[sd']) &= sd' \end{aligned}$$

This transformation puts a new root tag nm onto the source data in the forward transformation, and after the backward transformation, beside updating the source data, the state of x is also changed with its parameter nm replaced by nm'.

`XDistribute`: Let $x =$ `new XDistribute`(n), where n is a natural number.

$$\begin{aligned} x.\texttt{tranForward}(e) &= [e, ..., e] \\ &\quad (n \text{ copies of } e) \\ x.\texttt{tranBackward}(e, td') &= \texttt{merge}(e, td') \\ \text{where} & \\ td' &= [e'_1, ..., e'_n] \end{aligned}$$

In this definition, `merge` is an auxiliary function, which returns an updated source data by combining all modifications on all copies of the source data. If several copies contain different modifications at the same place, then these are conflict modifications. In this case, `merge` will raise an exception and the transformation is aborted. `XDistribute` is more general than `Dup` in [4], which is used to maintain data dependency relation in target documents. Moreover, by using `merge` function, `XDistribute` allows to update source data in a batch style rather than the interactive style adopted by `Dup`, so it is suitable for updating XML document in a network environment.

`XChildren`: Let $x =$ `new XChildren`() and $e = <tag>[e_1, .., e_n]$.

$$\begin{aligned} x.\texttt{tranForward}(e) &= [e_1, .., e_n] \\ x.\texttt{tranBackward}(e, [e'_1, .., e'_n]) &= <tag>[e'_1, .., e'_n] \end{aligned}$$

This transformation corresponds to the `child` axis in XPath. We will bidirectionalize XPath expressions using this transformation and some transformation combinators later. We also implement another commonly used axis, the `descendant` axis, in the class `XDescendant`.

5.2 Transformation Combinators

Transformation combinators are used to build new transformations from already defined transformations.

`XSeq`: Let $x =$ `new XSeq`$([x_1, ..., x_n])$.

$$\begin{aligned} x.\texttt{tranForward}(d_0) &= d_n \\ x.\texttt{tranBackward}(d_0, d'_n) &= d'_0 \\ \text{where} & \\ d_i &= x_i.\texttt{tranForward}(d_{i-1}) \\ d'_{i-1} &= x_i.\texttt{tranBackward}(d_{i-1}, d'_i) \\ & \quad (1 \le i \le n) \end{aligned}$$

`XSeq` takes a list of transformation objects as its argument. These argument transformations are applied sequentially in transformation. Note that the transformation x is updated if some of its argument transformations are updated during backward transformations.

`XMap`: Let $x =$ `new XMap`(x') and $sd = [e_1, ..., e_n]$.

$$\begin{aligned} x.\texttt{tranForward}(sd) &= td \\ x.\texttt{tranBackward}(sd, td') &= [e'_1, ..., e'_n] \\ \text{where} & \\ td &= td_1 + \cdots + td_n \\ td_i &= x'.\texttt{tranForward}(e_i) \\ (td'_1, .., td'_n) &= \texttt{split}(td', [|td_1|, ...|td_n|]) \\ e'_i &= x'.\texttt{tranBackward}(e_i, td'_i) \\ & \quad (1 \le i \le n) \end{aligned}$$

In this definition, the operator + is to concatenate two lists. In the backward transformation, the target data td' is divided into n sublists using the operator `split`, and the ith sublist has length $|td_i|$. Updating transformation itself is a bit complex for this transformation. Suppose that after backward transformation of td'_i $(1 \le i \le n)$, the

transformation object x' is updated to x'_i. Then, we can generate a new transformation element by using `merge`$(c', [c'_1, ..., c'_n])$, where c' and c'_i are the XML representations of x' and x'_i, respectively. This transformation element will be interpreted as a transformation object, and then used to replace the old object x'. Note that if `merge` raises an exception due to conflicts, the transformation will be aborted.

`XZip`: Let x = `new XZip`$([x_1, ..., x_n])$ and $e = <tag>[e_1, ..., e_n]$.

$$
\begin{array}{rcl}
x.\texttt{tranForward}(e) & = & <tag>[td] \\
x.\texttt{tranBackward}(e, d') & = & <tag'>[e'_1, ..., e'_n] \\
\text{where} & & \\
td & = & td_1 + \cdots + td_n \\
td_i & = & x_i.\texttt{tranForward}(e_i) \\
<tag'>[td'] & = & d' \\
(td'_1, .., td'_n) & = & \texttt{split}(td', [|td_1|, ...|td_n|]) \\
e'_i & = & x_i.\texttt{tranBackward}(e_i, td'_i) \\
 & & (1 \le i \le n)
\end{array}
$$

The actual implementation of `XZip` is more flexible than this definition. It allows the argument transformations of `XZip` and the contents of the source data e to have different length. If the former is longer, then the extra tail of the argument transformations are ignored; if the latter is longer, then the extra contents are processed by identity transformations.

`XIf`: Let x = `new XIf`$(pred, x_1, x_2)$.

$$
\begin{array}{rcl}
x.\texttt{tranForward}(sd) & = & x_1.\texttt{tranForward}(sd) \\
 & & \text{if } pred.\texttt{qualify}(sd) \\
 & = & x_2.\texttt{tranForward}(sd) \\
 & & \text{otherwise} \\
x.\texttt{tranBackward}(sd, td) & = & x_1.\texttt{tranBackward}(sd, td) \\
 & & \text{if } pred.\texttt{qualify}(sd) \\
 & = & x_2.\texttt{tranBackward}(sd, td) \\
 & & \text{otherwise}
\end{array}
$$

In this definition, *pred* is an object with interface `XPredicate`, which has a method `qualify` to judge whether the source data element satisfies some conditions. We have implemented several commonly used predicates, such as `XTrue`, `XLessThan`, `XGreaterThan`, `XEquals`, `XHasChild` and `XWithTag`, and three predicate operators `XAnd`, `XOr` and `XNot`. The meaning of each predicate is obvious. For example, `XWithTag(`*nm*`)` is to judge whether the source data is an element with the tag *nm*.

5.3 Bidirectional XPath

There are some transformations that need not to be defined primitively. Rather, they can be defined with the existing transformations. We give two examples in this section.

`XChildrenNm`: Let x = `new XChildrenNm`(nm).

$$
\begin{array}{rcl}
x.\texttt{tranForward}(e) & = & tran.\texttt{tranForward}(e) \\
x.\texttt{tranBackward}(e, td) & = & tran.\texttt{tranBackward}(e, td) \\
\text{where} & & \\
tran & = & \texttt{new XSeq}([x_1, x_2]) \\
x_1 & = & \texttt{new XChildren}() \\
x_2 & = & \texttt{new XMap}(x_3) \\
x_3 & = & \texttt{new XIf}(pred, x_4, x_5) \\
x_4 & = & \texttt{new XID}() \\
x_5 & = & \texttt{new XHide}() \\
pred & = & \texttt{new XWithTag}(nm)
\end{array}
$$

This transformation corresponds to the XPath step `child::`*nm*.

`XPathStep`: Let x = `new XPathStep`$(nm, pred)$.

$$
\begin{array}{rcl}
x.\texttt{tranForward}(e) & = & tran.\texttt{tranForward}(e) \\
x.\texttt{tranBackward}(e, td) & = & tran.\texttt{tranBackward}(e, td) \\
\text{where} & & \\
tran & = & \texttt{new XSeq}([x_1, x_2]) \\
x_1 & = & \texttt{new XChildrenNm}(nm) \\
x_2 & = & \texttt{new XMap}(x_3) \\
x_3 & = & \texttt{new XIf}(pred, x_4, x_5) \\
x_4 & = & \texttt{new XID}() \\
x_5 & = & \texttt{new XHide}()
\end{array}
$$

This transformation corresponds to the XPath step `child::`*nm*[*pred*]. It can be represented as follows in XML format:

```
<xpathstep>
  <name>nm<name> pred
</xpathstep>
```

With this transformation, the XPath expression $/nm_1[pred_1]/.../nm_n[pred_n]$ is encoded as the following bidirectional transformation:

```
<xseq>
  <xmap>
    <xpathstep>
      <name>nm1<name> pred1
    </xpathstep>
  </xmap>
  ...
  <xmap>
    <xpathstep>
      <name>nmn<name> predn
    </xpathstep>
  </xmap>
<xseq>
```

```
<lib>
  <name>TU Lib</name>
  <shelf>
    <category>Engineering</category>
    <cabinet>
      <book>
        <title>Data Structure</title>
        <author>Tom</author>
        <price>33</price>
        <year>2004</year>
        <publisher>
          <name>TU Press</name>
          <addr>US</addr>
        </publisher>
      </book>
      ......
    </cabinet>
    ......
  </shelf>
  <shelf>
    <category>Science</category>
    .......
  </shelf>
</lib>
```

Fig. 5 A Source XML Document

5.4 Degraded Bidirectional Transformation

`BiXJ` does not support modifications on the target data generated by non-invertible functions. That is, users should not modify these data, and even if they make some modifications, these modifications will not be reflected back. In BiXJ, we provide an abstract class `XActionNFun` for implementing non-invertible functions, which has implemented a trivial backward transformation by just returning the original source data. Hence, when implementing a non-invertible function, we just need to define a subclass of `XActionNFun` and implement this function in the forward transformation method.

In addition to implementing non-invertible functions, the abstract class `XActionNFun` can also be used to incorporate other existing XML transformation Java code into the bidirectional transformation framework if we do not care the backward transformations of these code.

6 Bidirectionalizing XQuery and XSLT

In order to test the expressiveness and the usability of the bidirectional transformations introduced above, we use them to bidirectionalize some typical examples in XQuery and XSLT, which are both popular and widely used XML processing languages.

```
<Books-of-Tom>{
  for $l in doc("lib.xml")/lib return
    for $s in $l/shelf[category="Engineering"]
    return
      for $c in $s/cabinet return
        for $b in $c/book
          where $b/author ="Tom" and $b/price<50
          return
            <book>{
              $b/title,
              $b/price,
              <press>
                {$b/publisher/name/text()}
              </press>
            }</book>
}</Books-of-Tom>
```

Fig. 6 An XQuery Expression

All examples in this section use an XML file "lib.xml" as the source document, which is partially listed in Figure 5. For this document, we assume that users are interesting in the engineering books written by Tom with a price less than 50 dollar.

6.1 Bidirectionalization of XQuery Expression

The structure of XQuery expressions generally takes the FLWR form. The interesting book information for users can be obtained by using the XQuery expression in Figure 6, which involves `for`, `where` and `return` expressions. This expression generates an element with the tag `Books-of-Tom`, which contains a list of `book` elements. And each `book` element has three child elements: `title`, `price` and `press`. The `press` element contains the publisher name.

The `BiXJ` script in Figure 7 implements the same transformation as the above XQuery expression. The whole script consists of a sequence of transformations wrapped by `xseq`. A simple way of writing this script is to finish this task step by step, that is, by adding a new transformation to the end of the existing script and looking at the transformation result, and then repeating this procedure until the expected result is obtained. However, it seems

```
<xseq>
  <xnewroot>Books-of-Tom</xnewroot>
  <xzip>
    <xseq>
      <xchildrennm>shelf</xchildrennm>
      <xmap>
        <xif>
          <xequals>
            <path>0</path>
            <value>Engineering</value>
          </xequals>
          <xid/> <xhide/>
        </xif>
      </xmap>
      <xmap>
        <xchildrennm>cabinet</xchildrennm>
      </xmap>
      <xmap>
        <xchildrennm>book</xchildrennm>
      </xmap>
      <xmap>
        <xif>
          <xand>
            <xequals>
              <path>1</path><value>Tom</value>
            </xequals>
            <xlessthan>
              <path>2</path><value>50</value>
            </xlessthan>
          </xand>
          <xid/><xhide/>
        </xif>
      </xmap>
      <xmap>
        <xseq>
          <xdistribute>3</xdistribute>
          <xnewroot>book</xnewroot>
          <xzip>
            <xchildrennm>title</xchildrennm>
            <xchildrennm>price</xchildrennm>
            <xseq>
              <xchildrennm>publisher</xchildrennm>
              <xchildrennm>name</xchildrennm>
              <xmodifyname>press</xmodifyname>
            </xseq>
          </xzip>
        </xseq>
      </xmap>
    </xseq>
  </xzip>
</xseq>
```

Fig. 7 Bidirectionalization of XQuery Expression

```
<xseq>
  <xnewroot>html</xnewroot>
  <xzip>
    <xseq>
      <xdistribute>2</xdistribute>
      <xzip>
        <xconst>
          <title>Books-of-Tom</title>
        </xconst>
        <xseq>
          <xnewroot>body</xnewroot>
          <xzip>
            <xseq>
              <xnewroot>table</xnewroot>
              <xzip>
                <xseq>
                  <xdistribute>2</xdistribute>
                  <xzip>
                    <xconst>
                      <tr>
                        <th>title</th>
                        <th>price</th>
                        <th>press</th>
                      </tr>
                    </xconst>
                    <xseq>x1  x2</xseq>
                  </xzip>
                </xseq>
              </xzip>
            </xseq>
          </xzip>
        </xseq>
      </xzip>
    </xseq>
  </xzip>
</xseq>
```

Fig. 9 Bidirectionalization of XSLT Expression

a tedious task. In the future, we would like to develop algorithms to translate the programs of high level XML processing languages into `BiXJ` code. In the following, we informally introduce some experiences learned when writing such script.

The XQuery expression in Figure 6 consists of two kinds of subexpressions: one is used to construct the target element and the other is to destruct the source element.

The first kind of expressions includes three element constructors for `Books-of-Tom`, `book` and `press` elements. They are encoded according to the following principle: If the source data already contains the content expected by a constructor, then

```
<xsl:stylesheet xmlns:xsl=...>
  <xsl:template match="/">
    <html>
      <title>Books-of-Tom</title>
      <body>
        <table>
          <tr>
            <th>title</th>
            <th>price</th>
            <th>press</th>
          </tr>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="/lib">
    <xsl:apply-templates select="shelf[category = 'Engineering']"/>
  </xsl:template>
  <xsl:template match="shelf">
    <xsl:apply-templates select="cabinet"/>
  </xsl:template>
  <xsl:template match="cabinet">
    <xsl:apply-templates
      select="book[author = 'Tom' and price < 50]"/>
  </xsl:template>
  <xsl:template  match="book">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="price"/></td>
      <td><xsl:value-of select="publisher/name"/></td>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Fig. 8 An XSLT Expression

we just modify the name of the source data with the element name specified in the constructor, otherwise we encode this constructor in the following form:

`<xnewroot>`*element name*`</newroot>`

`<xzip>`*code for constructing content*`</xzip>`

In Figure 7, the constructor for `press` element is encoded by simply modifying the name of the source data to `press`, and the other two constructors are encoded using the above form.

The second kind of expressions includes XPath expressions. An XPath expression is a sequence of path steps, and each step consists of an axis, a node test and a qualifier. To encode the child axis and name node test, the transformation `xchildrennm` is used, probably with the help of `xmap` to process an element list. The transformation `xmap` plays the similar role as the `for` clause in `XQuery`. The qualifier in an XPath step, such as `category = ''Engineeering''`, is encoded by an `xif` following the corresponding axis and node test. As a comparison, in the next section, the XPath expression in XSLT is encoded in the derived transformation `xpathstep`.

6.2 Bidirectionalization of XSLT

The style sheet of XSLT generally is made up of a list of templates, which are connected by `apply-templates`. The style sheet in Figure 8 generates the same interesting book information as the above XQuery expression. This style sheet includes five templates and transforms the data of interest into an HTML file.

The `BiXJ` code in Figure 9 implements the same transformation as the above XSLT style sheet. The code is divided into three parts for readability. The code in Figure 9 corresponds to the first template in Figure 8; the code x_1 in Figure 10 extracts the interesting books from the source document, corresponding to the second, third and fourth templates; the code x_2 in Figure 11 does the same thing as the last template, which is to construct table rows.

The bidirectionalizing procedure starts with the top template. When meeting with an `apply-templates` in a template, we put here the bidirectionalizing result of the applied template. For each template, we almost follow the same rules as used in bidirectionalizing XQuery expressions. The exception is that if an element constructor contains constant content, that is, it does not contain content computed by `apply-templates`, then we use `xconst` to construct this element directly. For example, the code for constructing the `title` element in Figure 9 belongs to this case.

7 Related Work

The `BiXJ` in this work takes a similar technical style as the bidirectional languages in [3][4]. As dis-

```
<xmap>
  <xpathstep>
    <name>shelf</name>
    <xequals>
      <path>0</path>
      <value>Engineering</value>
    </xequals>
  </xpathstep>
</xmap>
<xmap>
  <xpathstep>
    <name>cabinet</name>
    <xtrue />
  </xpathstep>
</xmap>
<xmap>
  <xpathstep>
    <name>book</name>
    <xand>
      <xequals>
        <path>1</path><value>Tom</value>
      </xequals>
      <xlessthan>
        <path>2</path><value>50</value>
      </xlessthan>
    </xand>
  </xpathstep>
</xmap>
```

Fig. 10 The BiXJ Code x_1

```
<xmap>
  <xseq>
    <xnewroot>tr</xnewroot>
    <xzip>
      <xseq>
        <xdistribute>3</xdistribute>
        <xzip>
          <xseq>
            <xchildrennm>title</xchildrennm>
            <xmodifyname>td</xmodifyname>
          </xseq>
          <xseq>
            <xchildrennm>price</xchildrennm>
            <xmodifyname>td</xmodifyname>
          </xseq>
          <xseq>
            <xchildrennm>publisher</xchildrennm>
            <xchildrennm>name</xchildrennm>
            <xmodifyname>td</xmodifyname>
          </xseq>
        </xzip>
      </xseq>
    </xzip>
  </xseq>
</xmap>
```

Fig. 11 The BiXJ Code x_2

cussed in Section 1, they have several limitations to be used as general XML transformation languages. In this work, BiXJ has addressed their limitations and is used for general purpose XML processing.

In the database area, there is also some work to do XQuery updating. For example, the work in [11] transforms updates on query tree into SQL updates, and then uses the database technology to update the database. Obviously, this technique is not suitable for updating native XML repositories. In addition, it cannot be used to update the view defined by XSLT, either.

The work [12] studies the problem of bidirectionalizing HaXML [13] and shows that any transformation in HaXML can be compiled into a bidirectional transformation. In work [14], the authors give an injective language Inv to implement view updating, and due to injectivity, so each program is invertible. However, they are still not used in general purpose XML processing. For example, they do not support bidirectional XPath.

8 Conclusion

In this paper, we solve the problem of view updating for general purpose XML processing. The proposed solution is a Java library BiXJ for bidirectional XML transformation. By this library, given a forward transformation, the backward transformation can be obtained for free. Hence, no extra efforts or separate mechanisms are needed for users to update source documents after target documents are modified. We have demonstrated the expressiveness and usability of BiXJ by bidirectionalizing the typical examples of two popular XML processing languages XQuery and XSLT.

In the future, we will develop algorithms that can translate XLST or XQuery expressions into the code of BiXJ automatically.

9 Acknowledgment

Thanks to the PSD project members in the University of Tokyo for stimulating discussion on this work. This work is partially supported by Comprehensive Development of e-Society Foundation Software Program of the Ministry of Education, Cul-

ture, Sports, Science and Technology, Japan. We are also grateful to the anonymous reviewers for their detailed and helpful comments and suggestions.

References

[1] W3C Draft. XSL Transformations (XSLT) Version 2.0 . http://www.w3.org/TR/xslt20/, 2005.

[2] W3C Draft. XML Query (XQuery) . http://www.w3.org/XML/Query, 2005.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

[4] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2004.

[5] W3C. The XML data model . http://www.w3.org/XML/Datamodel.html, 2005.

[6] W3C. XML Path Language (XPath) . http://www.w3.org/TR/xpath, 1999.

[7] Sun Developer Network (SDN). Java Technology and Web Services . http://java.sun.com/webservices.

[8] J. Hunter and B. McLaughlin. JDOM Project. http://www.jdom.org.

[9] D. Liu, Z. Hu, and M. Takeichi. An environment for maintaining computation dependency in XML documents. In *Proceedings of ACM Symposium on Document Engineering*, 2005.

[10] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 1998.

[11] V. Braganholo, S. Davidson, and C. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *Proceedings of International Conference on Very Large Databases (VLDB)*, 2004.

[12] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23:129–141, 2006.

[13] Malcolm Wallace and Colin Runciman. Haskell and XML: generic combinators or type-based translation? In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, 1999.

[14] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bidirectional updating. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, 2004.